Enhancing In-Network Computing Deployment via Collaboration Across Planes

Xiaoquan Zhang, Lin Cui, *Member, IEEE,* WaiMing Lau, Fung Po Tso, *Senior Member, IEEE*, Yuhui Deng, *Member, IEEE*, and Weijia Jia, *Fellow, IEEE*

***Abstract*—The new paradigm of In-network computing (INC) permits service computation to be executed within network paths, rather than solely on dedicated servers. Although the programmable data plane has showcased notable performance advantages for INC application deployments, its effectiveness is constrained by resource limitations, potentially impeding the expressiveness and scalability of these deployments. Conversely, delegating computational tasks to the control plane, supported by general-purpose servers with abundant resources, offers increased flexibility. Nonetheless, this strategy compromises efficiency to a considerable extent, particularly when the system operates under heavy load. To simultaneously exploit the efficiency of data plane and the flexibility of control plane, we propose *Carlo*, a cross-plane collaborative optimization framework to support the network-wide deployment of multiple INC applications across both the control and data plane. *Carlo* first analyzes resource requirements of various INC applications across different planes. It then establishes mathematical models for resource allocation in cross-plane and automatically generates solutions using proposed algorithms. We have implemented the prototype of *Carlo* on Intel Tofino ASIC switches and DPDK. Experimental results demonstrate that *Carlo* can effectively trade off between computation time and deployment performance while avoiding performance degradation.**

***Index Terms*—P4, QoS, Network function, In-network computing, Programmable data plane**

I. Introduction

THE emerging concept of in-network computing (INC) revolutionizes conventional cloud server-centric computing by allowing tasks to be offloaded partially or entirely to network components, capitalizing on their high-speed processing abilities [1]. INC applications cover not only emerging computing services (e.g., In-Cache [2]) but also conventional network functions (e.g., flow detection [3]). Emerging programmable data planes (PDP), such as the Intel Tofino switch ASIC [4], exhibit high-performance packet processing capabilities and enable line rate processing, improving Quality of Service (QoS) for INC applications [5].

Xiaoquan Zhang, Lin Cui, WaiMing Lau and Yuhui Deng are with the National & Local Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Provincial Key Laboratory of Data Security and Privacy Protection, College of Information Science and Technology, Jinan University, Guangzhou, 510632. China. Email: zhangxiaoquan547@gmail.com; tcuilin@jnu.edu.cn; raymondlau7176@gmail.com; tyhdeng@email.jnu.edu.cn.

Fung Po Tso is with the Department of Computer Science, Loughborough University, UK. Email: p.tso@lboro.ac.uk.

Weijia Jia is with Institute of Artificial Intelligence and Future Networks, Beijing Normal University (BNU Zhuhai), and BNBU, Zhuhai, China. Email: jiawj@bnu.edu.cn.

Corresponding author: Lin Cui

TABLE I: Comparison of devices across planes. Information is surveyed from prior works [11], [15] and product briefs [4].

Device	Throughput	Memory	Operation	Latency	Price
Server (control plane)	O(Gbps)	O(GB)	Complex	O(ms)	$3K
Switch (data plane)	O(Tbps)	O(MB)	Simple (bit shift, hash, etc.)	O(μs)	$10K

Implementing complex functions in PDP often involves trade-offs, leading to either degraded performance [6] or higher resource consumption [7], due to certain resource limitations presented in the programmable pipeline, such as Intel Tofino ASIC. Firstly, the available memory resources are substantially smaller (in the order of $\mathcal{O}(MB)$) compared to the $\mathcal{O}(GB)$ typically found in commodity servers. This restricts the deployment of stateful applications that require larger memory storage for network states (e.g., stateful NAT [8]). Secondly, the limited computing capability, primarily supporting addition and bit operations, hinders the implementation of complex functions.

Table I provides a comparison between PDP switches (data plane) and commodity servers (control plane). Considering the distinctive characteristics between the two planes, collaborative efforts across planes have been proven to be able to undoubtedly enhance application performance by leveraging the strengths themselves [9], [10]. For example, the high-performance execution of an in-cache application [2] relies on precise matching tables in the data plane, while the dynamic update strategy to maintain a high hit rate necessitates implementation in the control plane.

So far, the potential of cross-plane collaborative optimization remains largely unexplored. SyNAPSE [10] and CLIP [9] proposed frameworks to generate program code in both planes based on empirical deployment decisions only for an individual INC application. However, due to resource constraints of the data plane pipeline, it is inevitable to consider distributed deployment while optimizing overall performance for multiple INC applications. Although some existing research has explored the distributed deployment of multiple applications [11]–[14], these studies have not explored the potential benefits of cross-plane collaboration.

The development of INC applications presents a trend of more practical applications empowering sophisticated functionality, pursuing the design goal of balancing expressive-

ness, scalability, robustness, etc. [1]. It turns out that the intricate logic of these applications, such as packet processing of control flows and traffic modes, directly demands deployment solutions to be impacted by more factors (e.g., deployment cost and network performance). Some works have discussed offloading conventional NFV on PDP. They have either only allowed deploying an entire network function on a single plane [16], or enhanced the performance of table-based functions (e.g., NAT) by offloading specific logic (e.g., forwarding) on a single switch [17]. These works are not capable of responding to the complexity of such emerging INC applications.

In this paper, we propose *Carlo*[1] (Cross-plane Collaborative Solution) for the deployment of multiple INC applications via cross-plane collaboration. To our best knowledge, this is the first work to study cross-plane collaboration to support a wide diversity of INC applications deployment. *Carlo* provides a fine-grained solution that automatically decomposes INC applications into logic modules and chooses each logic module to deploy on appropriate planes and optimize performance. *Carlo* consists of three key components: application analyzer, deployment model, and optimization. To support deep analysis for a wide diversity of INC applications, the application analyzer decomposes the logic of each INC application as nodes by Control Flow Graph (CFG) for placement in the distributed data plane and identifies various traffic modes of INC applications to their required deployment resources in the control plane. The deployment model formulates an Integer Linear Programming (ILP) problem that enables the placement of nodes across different planes, optimizing deployment objectives while adhering to cross-plane resource constraints, thereby avoiding performance degradation or even deployment failure. The optimization module leverages proposed algorithms with varying time complexities to balance performance and search time. We conducted a testbed prototype based on P4 hardware switches (equipped with Intel Tofino ASIC) and servers with DPDK for experiments with various INC applications. Results of extensive experiments demonstrate that *Carlo* can optimize the deployment objective while avoiding performance degradation.

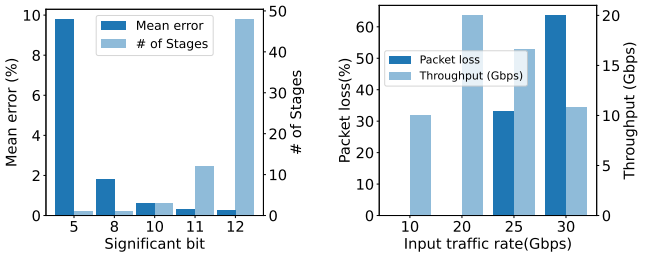
The rest of the paper is organized as follows. Section II gives background, motivation, and related works. Section III introduces the overview of *Carlo*. Section IV details the deep analysis of INC applications, Section V models the system and formulates the problem, and Section VI describes the two optimization algorithms. Section VII discusses the implementation, and Section VIII conducts experiments and analyzes their results. Section IX concludes this work.

II. Background and Motivation

A. Background

PISA (Protocol-Independent Switch Architecture) is a programmable match-action pipeline that aligns well with contemporary switching hardware. PISA incorporates several programmable components, namely the parser, pipeline, and deparser. The parser and deparser are responsible for extracting and reconstructing headers from network packets. The pipeline plays a crucial role in PISA as it allows for customization of application functionality by manipulating match-action tables (MAT). Computing and storage resources (e.g., SRAM, TCAM, and ALUs) are evenly divided into multiple stages within the pipeline (e.g., the Tofino 1 pipeline consists of 12 stages), and these resources are independently accessible. With the high-level language P4, users can manipulate these programmable components to tailor applications as required.

[1]A preliminary work appears as a conference paper published by 2024 IEEE International Conference on Computer Communications (INFOCOM) [18].



(a) Division operation in switches. (b) AES operation in servers.

Fig. 1: Motivating experiments with the implementation of two common operations.

INC applications. The primary goal of INC is to achieve significantly higher throughput and reduced latency by transferring specific computational tasks from end hosts to network devices like switches [1]. Due to the resource constraints of PDP, a complex operation must be either decomposed into small fractions among multiple stages or applied in lookup tables [19]. For example, inference of a decision tree can be encoded into multiple MATs by carefully splitting the feature spaces and mapping them [6]. Using lookup tables can achieve complex operations (e.g., division [20] and entropy [21]). However, these methods still face challenges in terms of scalability and accuracy.

B. Motivating experiments

To explore and understand how limitations in a single plane affect INC application deployments, the following experiments around two basic operations are conducted. The Division is a basic operation commonly used in many data plane applications, with accuracy ranging from a coarse-grained computation of flow rates for traffic control [20] and a high precision needed for in-network scientific computing workloads [7]. AES is the other significant operation widely used in security-oriented applications [22]. In the first experiment, we implemented a division operation in an Intel Tofino switch [4] using lookup tables. It can be observed from Fig. 1(a) that as the number of significant bits in the operands of the division operation increases, the accuracy also increases. Unsurprisingly, the number of stages required for the deployment of lookup tables also increases. Interestingly, the linear increase in significant bits incurs an exponentially higher deployment cost (e.g., stage usage), while the improvement in accuracy gains diminishes. In the second experiment, we built a topology where a sender server is connected to a receiver server

TABLE II: Summary of main related works.

Works	Multiple Apps	Optimi-zation	Collab-oration	Testbed
MTP [11]	✓	✓	✗	Tofino+DPDK
SPEED [24]	✓	✓	✗	Tofino
pSFC [12]	✓	✓	✗	Tofino
MrgRecmp4 [25]	✓	✓	✗	Tofino
ClickINC [14]	✓	✓	✗	Tofino+NIC
LightNF [16]	✓	✓	✗	Tofino
P4SFC [17]	✗	✗	✓	Tofino+DPDK
SyNAPSE [10]	✗	✗	✓	Tofino
CLIP [9]	✗	✗	✓	Tofino+DPDK
***Carlo* (ours)**	✓	✓	✓	Tofino+DPDK

via a 40Gbps fiber link. The sender generated packets with a size of 256 bytes ranging from 10Gbps to 30Gbps via Pktgen-DPDK [23]. The receiver collected AES packets and handled them through DPDK (Data Plane Development Kit) [23]. In Fig. 1(b), we can observe throughput degradation and severe packet loss when we increase the traffic rate of AES packets due to the server bottleneck.

These experiments demonstrate that the inherent constraints in a single plane may result in INC application deployments with inefficient resource allocation. This would produce sub-optimal solutions in deployment optimization.

C. Related works

Although many deployment frameworks in PDP have been proposed, none of these approaches are capable of optimization for multiple INC applications collaboratively in cross-plane. We categorize existing approaches into three groups: distributed deployment in PDP, cross-plane collaboration, and NFV offloading onto PDP.

Distributed deployment in PDP. Previous work has proposed frameworks to address the network-wide deployment for distributed PDP [11], [12], [14], [24], utilizing various graph tools to abstract network functions (e.g., table dependency graph [11], [24] and CFG [12]). MTP [11] focused on optimizing the placement of measurement functions based on the analysis of table dependency graphs, with the control plane exclusively responsible for handling analysis tasks generated by the data plane. SPEED [24] focuses on resource-efficient deployment of multiple P4 applications by merging identical table dependency graphs to reduce hardware resource usage. pSFC [12] proposed the concept of CFG for detailing more significant components for pipeline placement and corresponding table composition strategies for optimization of deployment. Unlike SPEED and pSFC, which only merge identical tables, MrgRecmp4 [25] opts to reconstruct pipelines for stateful service function chains (SFCs) with network-wide optimization. ClickINC [14] deployed modules of applications along multiple data plane paths, with programmable network elements (e.g., network interface cards, programmable switches, etc) processing the applications along the path. They have proven the effectiveness of function modulation in a deployment problem with multiple programmable targets.

Besides, recent studies have explored reinforcement learning (RL) to generate placement policies efficiently and improve

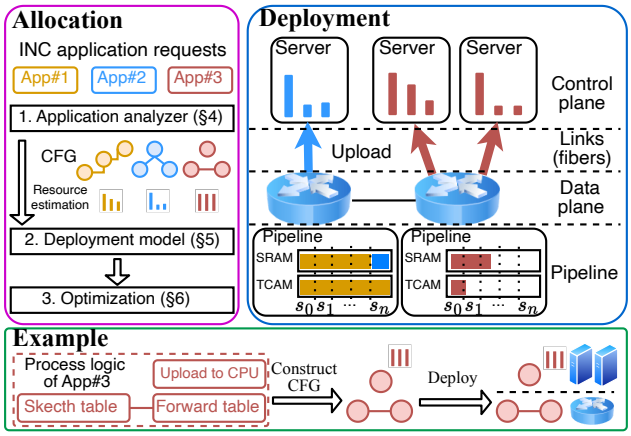


Fig. 2: Overview of architecture.

searchability for the deployment problem [26], [27]. Building on prior work, we analyze the traffic model of INC applications applied to the control plane. Based on this analysis, *Carlo* enhances the expressiveness of applications deployment and improves network-wide resource allocation through optimized placement strategies.

Cross-plane collaboration. Current research has discussed the advantages of cross-plane collaboration and proposed compilers to generate code in different planes [9], [10]. SyNAPSE [10] designed a compile language to simplify code design on a cross-plane. CLIP [9] proposed a framework to analyze and compile the P4 program code in a cross-platform. In comparison, *Carlo* extended the problem to network-wide deployments with multiple applications and demonstrated the effectiveness of cross-plane collaboration. Furthermore, utilizing the fine-grained application analysis method and optimization algorithm, *Carlo* can serve as a complementary framework to the above approaches by automatically generating optimized strategies for distributed deployment.

Offloading NFV onto PDP aims to offload specific *simple* logic on the data plane for conventional network function acceleration [16], [17]. LightNF [16] concluded the offloadability of conventional network functions and proposed an offloading framework to place them in either switches or servers. Similarly, P4SFC [17] differentiated the offloadability of network functions and generated both partial and full offloading strategies for SFCs. However, the current INC applications have chosen to deploy complex logic on the data plane [6], [7], [22], which comes at the cost of consuming more precious resources, causing a complicated tradeoff between performance and deployment costs. In short, *Carlo* enriches NFV offloading deployment optimization by supporting a wide variety of INC applications.

III. *Carlo* Overview

The key idea of *Carlo* is to automate the decomposition and deployment of INC application logic crossing both the data plane (e.g., switches with Tofino ASIC) and control plane (e.g., CPU-based servers).

A. Design challenges

We mainly face two challenges in designing *Carlo*. *The first challenge lies in accurately characterizing the properties of individual INC applications.* Understanding them helps in more accurately determining resource upper bounds to prevent performance degradation and compilation failure. To address this challenge, *Carlo* incorporates an application analyzer module, as shown in Fig. 2), to provide a general abstraction for fine-grained placement in the hardware pipeline and summarize the traffic mode for control plane processing.

The second challenge is that optimizing deployment objectives in cross-plane is NP-hard. The proposed algorithms should effectively generate collaborative policies to optimize deployment objectives while respecting constraints in distributed data planes and servers. In addition, to address scalability issues, algorithms must appropriately balance performance and execution time based on system requirements. *Carlo* includes a deployment model (Fig. 2) to model cross-plane constraints mathematically, and optimization modules to solve the tradeoff.

B. Carlo architecture

The overall architecture of Carlo and its workflow is shown in Fig. 2.

Deployment. *Carlo* allocates resources for deployment from a control plane (servers) and a data plane (pipeline resource) (the right part of Fig. 2). In the data plane, resource allocation must respect limitations in the PISA hardware pipeline to avoid compilation failure (Section II-A). In the control plane, a server needs to allocate resources to process the traffic of an INC application if the application is entirely deployed on the server. Meanwhile, some applications in the data plane would emit traffic to the control plane for analysis (e.g., analyze anomaly flows [28]). Thus, constraints of bandwidth and CPU resources for traffic processing should be considered to avoid performance degradation.

Allocation. To effectively allocate resources from cross-plane, *Carlo* proposes three modules (the left part of Fig. 2):

- *Application analyzer:* It analyzes the deployment resource requirements of each INC application by constructing its control flow graph (CFG), estimating pipeline resources or CPU/bandwidth if deployed in the data plane or the control plane, respectively.
- *Deployment model:* It builds a mathematical model to formulate cross-plane resource constraints, enabling application logic to be flexibly split and placed across different planes.
- *Optimization algorithms:* It searches for optimal deployment solutions that maximize objectives (e.g., throughput) while balancing algorithm performance and execution time.

Example. Fig. 2 shows the deployment workflow of Carlo for three INC applications. Specifically, in the application analyzer module, three CFGs are acquired to represent the three applications, as well as resource estimation in servers. For example, as shown in the bottom part of Fig. 2, App#3 includes three parts of process logic: sketch tables, forward tables, and uploading to the CPU for sketch analysis. Thus, three nodes are constructed within a CFG of App#3. The construction of the deployment model is based on the input of three applications and available resources in cross-plane (including three servers and two switches). As a result, the optimization module searches for a solution where App#1 nearly exhausts resources in the left pipeline, and both App#2 and #3 require both switch and server resources to process their traffic. In the example of App#3, sketch tables and forward tables are deployed in the data plane, and the operation of uploading sketch data to CPU for deep analysis is deployed in servers.

IV. Application analyzer

A. CFG abstraction of INC applications

CFG is a graph tool to provide deep analysis and fine-grained decomposition of critical components for a P4 program [12]. A directed acyclic graph $G = \{V, E\}$ can be used to abstract a given P4 program, where V is the set of nodes that refers to MAT or conditions in P4 programs, and E is the set of directed edges each of which describes the execution direction of two nodes. In Fig. 2, App#3 is decomposed into a CFG where each node represents a processing component, where edges connect two nodes (i.e., sketch and forward tables) to represent their execution sequence.

Dependency. Compilation violation occurs when two nodes having a dependency relationship are placed in the same stage [4]. For example, they mutually modify the same header fields. Thus, a dependency matrix D is defined to record relationships among different nodes in the CFG, where $D(n_i, n_j) = 1$ if the two nodes have a dependency.

Properties of CFG nodes. A CFG node possesses three attribute types within cross-plane. The first attribute is resource usage in the hardware pipeline, including SRAM, TCAM, and ALUs resources, which can be inferred by the calculation of the sizes of MAT and registers [13]. For instance, in App#3 (Fig. 2), the sketch table consumes SRAM and the forward table occupies TCAM in the pipeline. The second attribute is the maximum traffic to be processed on the server. The third attribute is the amount of CPU resources required to process received traffic on the server. Moreover, resource attributes of the same node are determined by which plane the node is placed in. For example, if the node is placed in the server, it would not consume any resources in the hardware pipeline. In the meantime, we also notice that nodes would have various traffic modes and CPU resource requirements when they are placed on different planes, such as the case in App#3, the analysis of sketch requires CPU resources.

B. Resource estimation in the control plane

To prevent performance degradation due to collaborative processing of network traffic beyond the capacity of the server, *Carlo* captures traffic modes of INC applications to compute the upper bound of traffic. Given an W as the input throughput of applications, the rate of maximum traffic to be processed in the server can be estimated (summarized in Table III).

TABLE III: Summary of traffic modes of INC applications.

Type	INC applications	Maximum traffic rate
Packet mirror	DDos detection [28]	W
Metadata mirror	DDos detection [28]	$W_p \cdot \sum_{f \in F} l_f$
Matching mirror	Traffic Engineering [29] Anomaly detection [29]	$W_m \cdot \lvert\mathbb{M}\rvert$
Query	Security detection [30] Monitor [31]	$\frac{\sum_{f \in F} l_f \cdot \delta_f}{t}$
Probabilistic	In-cache [2], [32]	$W \cdot (1 - \hat{\phi})$
Aggregating	In-band network telemetry [33], [34]	W_{int} $\theta \cdot W$
Bi-design	Complex operations [35]–[38] ML apps [3], [6], [39]	W 0

Algorithm 1: Traffic Mode Recognition

Input: G, the given CFG $G = \{V, E\}$
Output: TM, the set of traffic modes

```
1  for each n_i ∈ V do
2      if n_i performs generating traffic to CPU then
3          if n_i copies packets to CPU then
4              if CheckMirrorMode(n_i) then
5                  TM.add(n_i,CheckMirrorMode(n_i))
6          else if n_i forwards packets to CPU then
7              TM.add(n_i, Probabilistic)
8      else if CPU queries from n_i then
9          TM.add(n_i, Query)
10     else if n_i sends packets to the INT aggregator then
11         if CheckAggregatingMode(n_i) then
12             TM.add(n_i,CheckAggregatingMode(n_i))
13     else if n_i performs complex operations then
14         TM.add(n_i, Bi-design)
15 return TM
```

Algorithm 1 summarizes the detailed workflow to identify all traffic modes.

Mirror mode. Applications mirror packet copies with specific content to the control plane for analysis. **Packet mirror mode** directly mirrors the whole packet. Its maximum traffic rate equals the throughput of the INC application (W). **Metadata mirror mode** extracts packet headers to compute monitoring features ($f \in F$) for mirroring. Its maximum traffic rate can be calculated by the multiplication of total input throughput in terms of the number of packets (W_p) and the size of each packet, which equals the total bit width of features needed (l_f). Another mode, **matching mirror**, is to mirror all packets of a flow (e.g., an anomaly flow) matching in the MAT entry ($m \in \mathbb{M}$). We define W_m as the maximum traffic rate of the flow matching by m, and the maximum traffic rate of this mode equals the multiplication of the size of MAT ($|\mathbb{M}|$) and W_m. The three mirror modes can be identified via the mirror contents (function *CheckMirrorMode()* in line 5 of Algorithm 1).

Query mode. Some applications require the control plane to proactively and periodically query the flow information stored in the data plane for deep analysis (e.g., machine learning-based applications and monitors). Let δ_f be the number of monitored flows defined by the demands of the applications. The maximum traffic rate for this mode is equal to the total memory size occupied by all features divided by the time t needed to retrieve these features. For example, a register is used to save flow feature f, whose total memory size equals the multiplication of feature bit width l_f and the number of monitored flows δ_f.

Probabilistic mode. In this mode, whether traffic is forwarded to the control plane for processing is determined by a probabilistic value. The most typical application is in-network cache [2], in which the probabilistic value can be represented by the hit rate in the data plane. Assume the hit rate of the given MAT table is $\hat{\phi}$, then the miss rate is $1 - \hat{\phi}$. Thus, the maximum traffic rate can be $F \cdot (1 - \hat{\phi})$. It is noted that the hit rate is an estimated value, and it varies due to network dynamics. In our experiments, we identified the lower bound of $\hat{\phi}$ by analysis of the experimental dataset.

Aggregating mode. INT (In-band Network Telemetry) applications require generating and aggregating INT packets that are processed in the control plane. The packet generation mainly includes two patterns, i.e., passive and proactive. The first is to sample normal traffic with a sample rate (θ), whose maximum traffic rate is $\theta \cdot W$. The second is to proactively generate probe packets with a fixed rate from an INT generator [33], whose maximum traffic rate is the fixed rate W_{int}. Function *CheckAggregatingMode()* (line 12 of Algorithm 1) checks the two different ways of packet generation.

Bi-design mode. Implementing complex functionality in the pipeline usually consumes more scarce resources. For example, entropy operations [37] need to occupy 20 stages in the Tofino 2 ASIC pipeline. *Carlo* allows users to make the trade-off between deployment cost and performance. Specifically, applications can be fully deployed in the control plane, where the maximum flow rate of the mode equals the application's maximum processing rate, or entirely offloaded in the data plane, in which no traffic is forwarded to the control plane.

Examples. The second attribute of CFG nodes can be estimated by capturing the traffic modes of applications. For example, an in-network machine learning (ML) application can be entirely placed in servers or the data plane (*Bi-design*). It can also offload feature statistics in the data plane to improve packet forwarding throughput (*Query* or *Mirror*).

CPU resource is calculated by the number of cycles in *Carlo* to identify resource requirements of INC applications for processing traffic in servers. To estimate that of each INC application, *Carlo* performs benchmark experiments to inject traffic and uses *perf* [40] to measure their CPU cycles.

V. Deployment Model

The deployment problem can be formulated to an ILP model, where each node ($n_i \in V$) of a compound CFG G that consists of multiple INC applications (i.e., each subgraph represents an application) is placed and allocated resources on cross-plane while respecting the resource constraints. Specifically, node attributes change depending on their deployment

TABLE IV: Notation of symbols.

Notation	Description
b_i^c	Maximum traffic rate of node n_i in the server c
B^c	Bandwidth between server c and its connecting switch
C	Set of servers $c \in C$
$D(n_i, n_j)$	Dependency between node n_i and n_j
G	Control flow graph, $G = \{V, E\}$, including node set $n_i, n_j \in V$ and edge set E
I_i, O_i	Input and output stage for a node n_i
P	Set of programmable switches $sw \in P$
R_i^m	Type m of resource requirement for node i
S	Set of all stages of switches in P
u_i^c	Maximum CPU utilization of node n_i in servers
U^c	Maximum CPU utilization of server c
V	Node set including two sets of nodes only being placed in switches (V_{sw}) and servers (V_c)
z_{sw}	Whether switch sw has node(s) to be placed
z_c	Whether server c has node(s) to be processed
μ^m	Limited type m of resources in each stage
ϵ_{sw},ϵ_c	Deployment cost for the switch sw and the server c
Variables	
$Q_i^{s,m}$	Type m of allocated resource for node i in stage s
x_i^s	Whether node n_i is placed in stage s
y_i^c	Whether node n_i is placed in server c

location. For example, if a node is placed in the data plane, it consumes pipeline resources and may generate traffic to servers. If it is placed in servers, it consumes no pipeline resources.

The network elements consist of a set of programmable switches ($sw \in P$) and a set of servers as the control plane ($c \in C$). Pipeline resources of a switch are evenly divided and represented in the form of the stage ($s \in S$), where S refers to the set of all stages of switches in P. It is noted that we don't consider latency between the data and control plane because we use fiber to link them, whose latency is far lower than that in servers for processing (e.g., 6*ns* of serialization delay in IEEE standard 802.3bm [41]).

Binary variables. x_i^s and y_i^c, are used to denote whether the node n_i is placed in stage s and is placed in the server c, respectively. z_{sw} and z_c, refer to whether switch sw and server c have been placed nodes, respectively.

Optimization constraints include CFG node placement and resource limitations in cross-plane.

(1) Constraints of node placement. Each node n_i must be assigned to one location, either in stages or in servers:

$$\forall n_i \in V : \bigvee_{s \in S} x_i^s + \bigvee_{c \in C} y_i^c = 1. \tag{1}$$

Moreover, a node can be placed among multiple stages in the data plane :

$$\forall n_i \in V : \bigvee_{s \in S} x_i^s = 1 \implies \sum_{s \in S} x_i^s \geq 1. \tag{2}$$

Besides, nodes that must be placed in servers ($n_i \in V_c$) or switches ($n_i \in V_{sw}$) must be assigned to the correct plane:

$$\forall n_i \in V_{sw} : \bigvee_{s \in S} x_i^s = 1; \forall n_i \in V_c : \bigvee_{c \in C} y_i^c = 1. \tag{3}$$

A node may consume different types of pipeline resources $M = \{SRAM, TCAM, ALU\}$ for computation and storage. An integer variable $Q_i^{s,m}$ represents the allocated resources of type $m \in M$ for node n_i in stage s. All resources of each node must be allocated completely:

$$\forall n_i \in V, \forall m \in M : \bigvee_{s \in S} x_i^s = 1 \implies \sum_{s \in S} Q_i^{s,m} = R_i^m, \tag{4}$$

where R_i^m refers to the requirement for resource type m of node n_i.

(2) Constraints in the hardware pipeline. Since each stage in the hardware has limited resources μ^m, the total allocation of resources of nodes in a stage must not exceed its capacity:

$$\forall s \in S, \forall m \in M : \sum_{n_i \in V} Q_i^{s,m} \cdot x_i^s \leq \mu^m. \tag{5}$$

If two nodes have a dependency (i.e., $D(n_i, n_j) = 1$), the output stage of node n_i must precede the input stage of node n_j.

$$\forall n_i, n_j \in V : D(n_i, n_j) = 1 \implies O_i < I_j, \tag{6}$$

where I_i and O_i are defined as the input and output stages of a node n_i.

(3) Constraints in the control plane. The accumulation of traffic rates of INC applications should not exceed the capacity of the link between the switch and the server:

$$\forall c \in C : \sum_{n_i \in V} b_i^c \leq B_c, \tag{7}$$

where b_i^c refers to the maximum traffic rate of node i. It is noted that to conserve server resource consumption, *Carlo* allows traffic sharing among multiple applications to apply the same tasks (e.g., multiple INC applications conduct the traffic mode of *packet mirror*). The same type of uploading traffic among applications would not accumulate.

For a server c, its capacity of CPU resources should be higher than the total CPU resources required by all nodes that it processes:

$$\forall c \in C : \sum_{n_i \in V} u_i^c \leq U_c, \tag{8}$$

where u_i^c refers to the maximum CPU utilization of node n_i required in servers.

Objectives. *Carlo* provides two common optimization objectives for model optimization separately. The first objective (#1) is to maximize overall throughput. It can be beneficial by minimizing the amount of traffic needed to be forwarded to servers:

$$\min(\sum_{c \in C} \sum_{n_i \in V} b_i^c \cdot y_i^c). \tag{9}$$

Network providers need to consider deployment costs to increase profits. Thus, the second objective (#2) is to minimize deployment costs:

$$\min(\epsilon_{sw} \cdot \sum_{sw \in P} z_{sw} + \epsilon_c \cdot \sum_{c \in C} z_c), \tag{10}$$

where ϵ_{sw} and ϵ_c refer to the deployment cost for the switch and the server.

NP hardness. The above problem is NP-hard. The NP-hardness can be proven by considering a special case of the problem. If we nullify the control plane resources, the problem becomes a deployment problem in PDP, which is an NP-hard problem [12], [13], [24].

Algorithm 2: Node-ranking heuristic search

Input: G, the given CFG $G = \{V, E\}$; D, the dependency matrix of G; *Topo*, the given topology; *Obj*, the objective of search
Output: *rel*, the deployment result

```
1  /* Sorting nodes∈ V for placement   */
2  V.Normalization(G,D)
3  V.Ranking(Obj)
4  if Obj is MinCost then
5      /* Prioritize placing in control
          plane to minimize cost        */
6      for each n_i ∈ V do
7          if n_i ∈ V_cp and SearchInCP(n_i, Topo) then
8              rel.add(SearchInCP(n_i, Topo))
9          else
10             rel.add(SearchInDP(n_i, Topo))
11 else
12     /* Prioritize placing in data plane
          to maximize throughput        */
13     for each n_i ∈ V do
14         if SearchInDP(n_i, Topo) then
15             rel.add(SearchInDP(n_i, Topo))
16         else
17             rel.add(SearchInCP(n_i, Topo))
```

VI. Optimization algorithms

The solution space of the deployment problem equals the exponential of the number of CFG nodes to the sum of total stages and servers, i.e., $(|S| + |C|)^{|V|}$ where $|S|$, $|C|$, and $|V|$ are the size of total stages, servers, and CFG nodes. For example, $|S|$ equals the multiplication of the number of available switches and stages per switch (e.g., 12 stages in Intel Tofino 1), and $|V|$ equals the multiplication of the number of applications and nodes per application. Thus, the scalability issue becomes prominent in optimal solvers (e.g., Gurobi) as the number of applications and switches increases. *Carlo* proposes two algorithms to trade off between performance and execution time.

A. Node-ranking heuristic search (NRHS) algorithm

The core idea of Algorithm 2 focuses on prioritizing nodes of a CFG and deploying them in cross-plane. First, it ranks the nodes of a given CFG based on different attributes (e.g., required resources and dependency relationships), and then incrementally searches and places each node while respecting resource constraints.

Node ranking. Sorting to determine which node can be placed earlier is important since it can affect the quality of placement [13]. For example, nodes with large resource requirements could be placed earlier to avoid insufficient remaining available resources. Given the placement request of a CFG, its nodes can be evaluated one by one using their attributes as indicators, including requirements of the pipeline, CPU resources, traffic amounts, etc. Therefore, *Carlo* uses min-max normalization to rescale each indicator in the range $[0, 1]$:

$$\kappa_i^k = \frac{\lambda_i^k - \lambda_{min}^k}{\lambda_{max}^k - \lambda_{min}^k}, \tag{11}$$

where λ_i^k is the indicator of type k of node n_i. Afterward, different optimization objectives not only determine how to design placement policies but also decide distinct metrics that affect the placement. A weighted sum of normalized metrics ε is used to sort the nodes from large to small (line 2∼3):

$$\forall n_i \in V : \varepsilon_i = \sum_{k \in Metrics} \omega^k \cdot \kappa_i^k, \tag{12}$$

where ω is the metric weight varying in deployment objectives.

Incremental placement. Next, the algorithm applies policies of incremental placement to place sorting nodes in sequence according to different objectives. To minimize cost, the algorithm prefers to place more nodes in V_{cp} as possible in the control plane, and with the other objective, on the contrary, the algorithm priorities to place nodes in the hardware pipeline to maximum throughput (line 4∼17 of Algorithm 2). Function *SearchInCP()* searches available resources in the server whose linking switch has been used and allocates for placing node n_i. If it succeeds, it returns the server ID; otherwise, it returns false. Function *SearchInDP()* searches the switch resources for placement of n_i.

Time complexity of this algorithm is $\mathcal{O}(|V|(|S| + |C|))$, where $|V|$, $|S|$, and $|C|$ refer to the number of CFG nodes, stages, and servers in the topology. In the worst case, each node must check the resources of all stages and servers for placement.

B. Policy gradient (PG) algorithm

The NRHS algorithm may get stuck in sub-optimal results due to its fixed searching policies. To enhance the algorithm performance, the policy gradient algorithm is proposed to automate studying deployment policies and generating deployment solutions within an acceptable time.

1) Reinforcement learning architecture: We build a reinforcement learning (RL) architecture in which the policy gradient algorithm is implemented in an RL agent to study deployment strategies via interaction with the RL environment (shown in Fig. 3). The environment is built to completely emulate the deployment process in both the data and control planes. At each time step $t \in T$, the agent obtains the current deployment state and samples an action a (i.e., a placing location in switches or servers) for deploying a CFG node, where the action mask temporarily masks out invalid locations to place. A reward r_t is obtained after executing the action, and the state is transferred to the next state. The objective of learning is to maximize the estimation of discounted accumulative rewards $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$, where γ is the discount factor.

State representation. The state is encoded by a feature vector that includes resources in cross-plane and placement conditions of the given CFG. The remaining resources usage in cross-plane is identified, including bandwidth and CPU

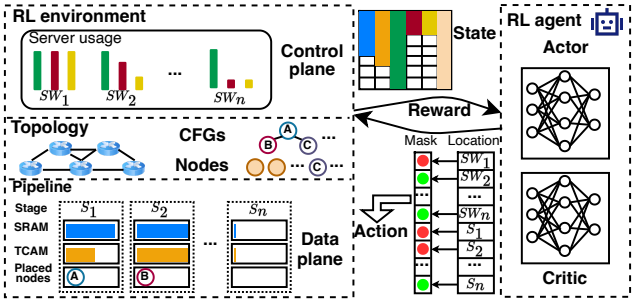


Fig. 3: Design of the RL architecture. At this step, nodes A and B have already been placed, and node C is currently under deployment. The state is encoded into a vector based on resources in cross-plane and placement conditions of the given CFG. Then the state vector is fed into Actor network to generate an action vector, where each element indicates the probability of selecting a specific location in cross-plane. A masking mechanism is applied to filter out invalid options, and the final action is sampled from the masked vector to determine a valid deployment location. Finally, a reward is calculated to guide future decisions.

resources of each server in the control plane, and storage (e.g., SRAM, TCAM) and computation resources (e.g., ALUs) in the pipeline of each switch in the data plane. Besides, previous placement details of the given CFG are also recorded in the feature vector, as well as attributes of the incoming placing node. Eventually, all features are flattened as a fixed-size matrix, and 0 is filled in the vacuum of the matrix. As shown in Fig. 3, different colored blocks in the state vector represent different types of features.

Action space. An action is defined as placing a CFG node in a certain location (i.e., a stage of the switch pipeline or a server) with resource allocation. However, the action space is enormously enlarged when it comes to resource allocation in the pipeline of switches. For example, since TCAM memory is split into tens of portions per stage in the Tofino hardware pipeline [4], the completion of one CFG node placement may contain multiple repeated actions in the same stage for TCAM allocation if one portion of TCAM is allocated per action. It turns out to be a longer period of policy convergence. To eliminate the problem, a greedy allocation mode is set for placement. Specifically, a node allocates all resources it needs in the placing location for placement; however, if the remaining resources in that location are fewer than its needs, it would be placed in another location at the next time step until it allocates sufficient resources. At each time step, only one placement location is picked as an action. Thus, an action vector is used to record the selected possibility of each location, whose index is the placement location and value is its probability. Subsequently, the RL agent samples one action from the action vector per time step.

Reward shaping. To accelerate the convergence of placement policies learning, intermediate rewards (e.g., positive and negative rewards) are generated at each time step to encourage the RL agent learning for higher accumulated rewards. By incorporating intermediate rewards, the RL agent receives continuous learning signals, which improves training stability and accelerates policy convergence, especially in environments where successful deployments are difficult to achieve in the early stages. This design is effective when CFGs have a larger number of nodes.

We define two types of intermediate rewards: (1) *Positive rewards* are given when a node is successfully placed, which include a reward (e.g., 0.5) assigned when reusing existing pipeline stages or servers, as well as a reward (e.g., 1) assigned when all nodes of a CFG have been placed successfully. (2) *Negative rewards* are used to penalize suboptimal or failed placements, oppositely. The negative rewards include a lower reward (e.g., 0.1) if a new stage or server is used to place, as well as a larger penalty (e.g., 10) assigned if the deployment fails due to constraint violation. These intermediate rewards are aggregated over time to form the final reward. Their relative magnitudes can be tuned to reflect different optimization goals via weight parameters. For instance, if minimizing resource usage is prioritized, weighting penalties for using new servers can be increased.

Search speedup. Due to strict constraints on the switch hardware and server resources, the deployment is easy to terminate because of constraint violations. This leads to the ineffectiveness of policy learning. To mitigate the negative impact, a masking mechanism is designed to cover those invalid actions. First, invalid deployment actions may occur when selected locations have exhausted their available resources and can no longer support additional placements. Second, invalid actions will lead to dependency violations between two nodes. The masking mechanism needs to consider whether there are dependency restrictions between deployed nodes and the current placing node in terms of deployment locations (constraint of Equation 6). A mask vector is used to set invalid locations aligned to the action vector with a negative *inf* value. By doing so, the agent can not sample those invalid locations from the action vector.

Example. In Fig. 3, three locations (i.e., SW_1, S_1, and S_2) are invalid when placing node C. The reasons are as following: 1) nodes A and C have dependency relationship and nodes A has been placed S_1, the placement of S_1 will break the dependency constrain; 2) resources in stage S_2 are depleted; 3) CPU and bandwidth resources in the server SW_1 are insufficient to place.

2) Policy gradient algorithm: The training algorithm (Algorithm 3) builds two neural networks, where Actor network is responsible for generating actions and Critic network evaluates the current environment. The algorithm trains the agent in two steps for each epoch. In the first step, the agent consistently interacts with the environment (line 4∼20) and saves historical deployment procedure (i.e., trajectory) in the *Buffer*. Specifically, the agent first selects a CFG node and samples an action from Actor network (line 6∼13), and then evaluates the current deployment in Critic network and executes the sampling action (line 14∼17).

In the second phase, the RL agent updates parameters in Actor and Critic networks using policy gradient loss computing in the *Buffer*. In Actor network, the gradient loss can be

Algorithm 3: Policy gradient training algorithm

Input: G, *Topo*, the given CFG and topology
g, *topo* $\leftarrow$ *Env_initialize*(G, *Topo*)
Initialize: θ_π, $\theta_\mathbb{V}$, parameters of Actor and Critic
for *epoch in Epochs* **do**
  Buf_initialize(*Buffer*)
  while *Epoch_flag* $\neq$ *True* **do**
    /* Get a node to be place */
    $n \leftarrow$ *Get_CFG_node*(g)
    /* Obatin state */
    State $\leftarrow$ *Get_Env_State*(g,*topo*)
    /* Masking invalid actions */
    log $p \leftarrow$ *Maskout* ($\pi(a|State, n; \theta_\pi)$)
    /* Select an action */
    $act \leftarrow$ *Sample*(log p)
    /* Get value from Critic */
    $v \leftarrow \mathbb{V}$(*topo*, n;$\theta_\mathbb{V}$)
    /* Apply the action */
    $g, topo, reward \leftarrow$ *Action*(g,*topo*,*act*)
    Buffer.add(log p, *act*, v, *reward*, *topo*, n)
    if *Trajectory_flag* == *True* **then**
      g, *topo* $\leftarrow$ *Env_initialize*(G, *topo*)
  /* Policy training */
  Compute loss via *Buffer* and update θ_π and $\theta_\mathbb{V}$ for Actor and Critic networks, respectively

TABLE V: *Experimental settings*

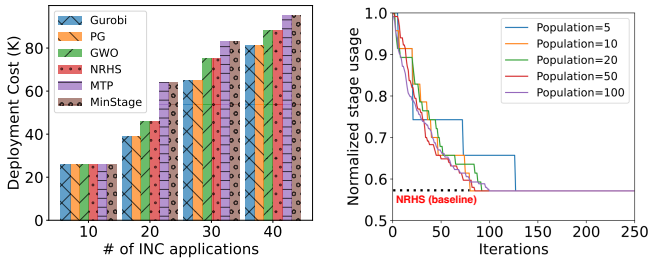
Parameter name	Value
Sample rate θ	$0.1 \sim 0.5$
Hit rate $\hat{\phi}$	$0.1 \sim 0.5$
Input throughput per application	$1 \sim 40$Gbps
Epoch	1024
Discount factor	0.99
GAE lambda	0.97
MLP layer	64×64, 128×128, 256×256
GWO population	5, 10, 20, 50, 100

calculated by the mean error between GAE-Lambda advantage estimate and the logit of the corresponding sampled action (i.e., *log p* across the epoch). In Critic network, the mean square error between the reward-to-go and the output of Critic network across the epoch, where the reward-to-go is computed by utilizing the discount factor to intermediate rewards.

Complexity Analysis. The computational complexity required for training the Actor and Critic network is $\mathcal{O}(RH + H^2 + HL)$ and $\mathcal{O}((R+L)H + H^2 + H)$, where R refers to the dimensionality of the state space, H refers to the number of neurons in two hidden layers in Actor and Critic, and L refers to the dimensionality of the action space. Thus, the overall time complexity of the algorithm is $\mathcal{O}(\mathcal{E}T(RH + H^2 + HL + (R + L)H + H^2 + H))$, where $\mathcal{E}$ represents the number of epochs and T represents the number of steps in each epoch.

VII. Implementation

We develop a testbed prototype targeting Intel Tofino ASIC switches and servers equipped with DPDK [23] for high-throughput packet processing. The p4i compiler [4] is used to parse P4$_{16}$ programs and generate a corresponding CFG for each application.



(a) Comparison of deployment cost. (b) Performance of heuristics.

Fig. 4: Comparison of deployment algorithms.

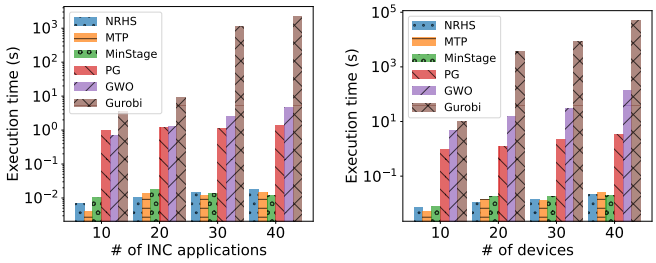
In the data plane, an *appID* is attached to each packet to identify the corresponding INC application. On each switch, a lightweight MAT is deployed to determine whether a packet needs to be redirected to the control plane. To support cross-switch deployment, metadata fields are reserved in packet headers to carry intermediate processing states with small additional overhead [24].

Dynamic deployment. When a new INC application request arrives, *Carlo* searches switches and servers with available resources for deployment. This allows for minimizing disruptions to the current deployment.

INC Applications. We evaluate *Carlo* using multiple INC applications introduced in Section IV-B. The application logic is partitioned into modules following the method proposed in CLIP [9], with control plane modules implemented in C (for DPDK) and data plane modules in P4 (for Tofino). Application-specific parameters, including sampling rate θ and cache hit rate $\hat{\phi}$, are varied in the range of 0.1 to 0.5. The two rates affect the amount of network traffic to be uploaded to CPUs for processing. In addition, anomaly flow entries from a public dataset are injected into MATs to emulate real-world traffic patterns. Detailed parameter settings are listed in Table V.

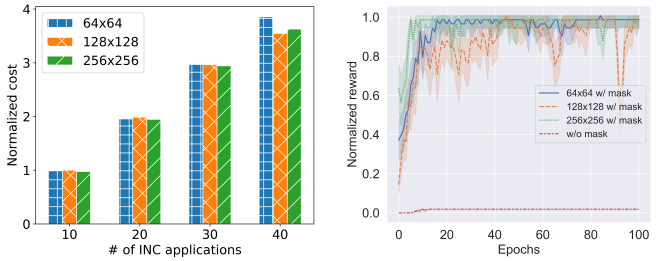
Optimization. Gurobi [42] is used to solve the ILP-based optimization model. The policy gradient algorithm is implemented using the SpinningUp framework [43] and trained on an NVIDIA Tesla V100 32GB GPU, where multi-layer perceptrons (MLPs) are used to construct the actor and critic networks. Training parameters are also summarized in Table V.

We implement the Grey Wolf Optimizer (GWO) to explore the deployment space without relying on handcrafted heuristics. The algorithm maintains a population of candidate solutions, where each "wolf" represents a potential deployment strategy. By iteratively evaluating and updating the population, GWO effectively searches for optimal or near-optimal solutions. A larger population generally improves search coverage but increases computational overhead.



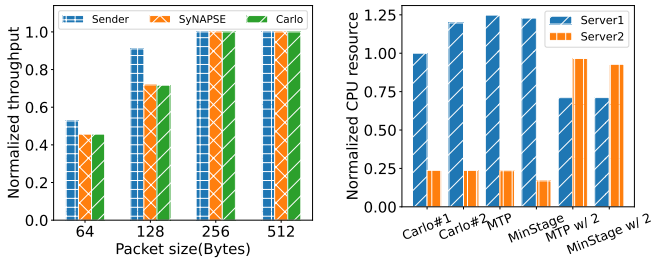
(a) Time w.r.t. # of INC applications. (b) Time w.r.t. # of devices.

Fig. 5: Comparison of execution time.

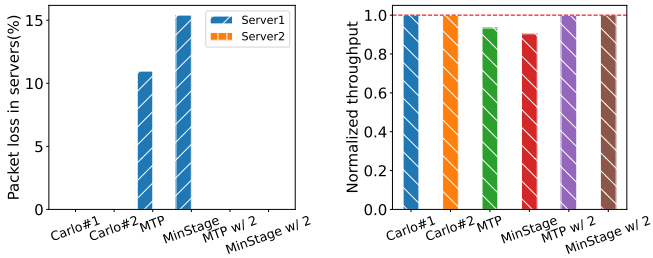


(a) Impact on training cost. (b) Impact on training reward.

Fig. 6: Performance of PG algorithm.



(a) NAT comparison (b) CPU resource



(c) Packet loss in servers (d) Throughput

Fig. 7: Performance in testbed experiments.

VIII. Evaluation

A. Experimental settings

Testbed. The testbed for experiment is built with two P4 hardware switches (Flnet S9180-32X with Intel Tofino ASIC). In the data plane, devices are connected in a linear topology, with two switches positioned at the center and each connected to a server. Additional servers are used as senders and a receiver. Senders use Pktgen-DPDK [23] to generate flows with 256-byte packets processing in monitoring applications (e.g., DDoS detection [28] and INT [33]) and compute tasks (e.g., AES [22] and ChaCha [44]), and *Tcpreplay* to replay the real dataset CIC-IDS2017 [45] processing in In-cache [2], ML applications [6], etc., generating a total of 40Gbps traffic rate. Each server is equipped with a dual Intel Silver 4210R CPU with 10 cores and a Mellanox ConnectX-3 40GbE Network Interface Card, and runs DPDK for packet processing. All servers are connected by the switch via 40Gbps fiber (40GBASE-SR4) links.

Comparison schemes. MTP [11] is a framework designed for deploying measurement tasks in PDP. MinStage [13] introduced a set of heuristics aimed at minimizing deployment costs within the hardware pipeline. However, both approaches focus solely on the data plane and do not consider cross-plane deployment.

B. Experimental Results

(Exp#1) Deployment cost of *Carlo*. We first evaluated the performance of *Carlo*'s algorithms on the objective of minimizing cost, and compared with other methods. Switch and server costs are set according to Table I. We randomly selected 10 to 40 INC applications for deployment, each with an input throughput ranging from 1∼40Gbps. As shown in Fig. 4(a), the PG algorithm achieved the same performance as Gurobi, indicating that it can be a near-optimal solution. On the other hand, the GWO and NRHS heuristic algorithms exhibited performance bottlenecks as the number of INC applications increased. However, MTP and MinStage are only deployed in the data plane, requiring more resources to meet the resource requirements, especially when deploying complicated applications. In conclusion, *Carlo* decreases by an average of 18.7% compared with the two methods.

(Exp#2) Performance of heuristics. In this experiment, we compared the performance of two heuristic algorithms (i.e., GWO and NRHS) in the deployment of 30 applications. Due to the stringent placement constraints in the data plane, measuring stage usage in the pipeline can evaluate the effectiveness of the two heuristics. The manually tuned strategy primarily depends on the specific inputs for the pipeline placement. In contrast, GWO can automatically adjust parameters to optimize placement performance based on given initial placement values. Therefore, to evaluate the GWO algorithm, we set different populations and an initial value with more stages of usage. As shown in the experiment (Fig. 4(b)), with an increase in the number of iterations, a more stable and rapid convergence appears. Ultimately, all figures converge to the deployment results of NRHS. However, the complexity of GWO could surge as the population and epoch numbers of GWO increase. Therefore, this experiment demonstrates that the NRHS algorithm can ensure deployment performance with low complexity.

(Exp#3) Execution time of *Carlo*. We measured the execution time of different algorithms in *Carlo* under two settings: varying the number of INC applications from 10 to 40 to evaluate impact on execution time (Fig. 5(a)), and varying the number of devices within the same range (Fig. 5(b)). As

expected, Gurobi identified the optimal solution but encountered scalability limitations, with the search time increasing exponentially as the number of devices grew. After a period of offline training, the PG algorithm achieved approximate solutions within a short runtime. For GWO, we fixed the number of generations and search iterations, but it also faced scalability challenges. Finally, heuristic algorithms (i.e., NRHS, MTP, and MinStage) can obtain feasible solutions in the shortest time due to their smaller search complexity.

(Exp#4) Performance of PG algorithm. We evaluated the effects of different MLP sizes on deployment cost with objective #2. In Fig. 6(a), the smallest neural network achieved a high cost in deployment requests. For example, at 40 applications, the cost of the 64×64 network is 8.7% and 6.3% higher than those of the 128×128 and 256×256 networks. This indicates that the smallest network was unable to model complex placement relationships effectively under larger-scale deployment scenarios.

We observed rewards in different sizes of MLP with and without the masking mechanism. Results in Fig. 6(b) show that the MLP with the largest hidden size has a faster convergence speed for the number of epochs because it can be trained more efficiently to capture the more complicated relationships. In contrast, without the masking mechanism, the agent struggles to learn effective policies due to the complexity of the deployment task and the high likelihood of failure.

(Exp#5) Performance in testbed experiments. In the first testbed experiment, we deployed one NAT application to compare Carlo with SyNAPSE [10], a cross-plane deployment for the individual application. In NAT, to prevent register space overflow in the switch leading to hash collisions, a portion of the traffic was forwarded to the controller for processing (we set hit rate $\hat{\phi}$ to 0.5). The experiment used a single server as the control plane to process forwarded packets and specified different packet sizes (e.g., 64, 128, 256, and 512 Bytes) at the other server as the *Sender*, and measured the throughput of the application, which is defined as the total amount of traffic processed successfully per second. As shown in Fig. 7(a), the server becomes the bottleneck that affects the processing rate of NAT packets in the control plane and the capacity of the packet generator in *Sender* when it sends packets with small sizes. Besides, Both methods achieve very close processing performance due to their similar deployment strategies, which tend to be deployed in the data plane to maximize throughput.

We then conducted testbed experiments for comparison with the other two methods for multiple applications deployments (shown in Fig. 7), where *Carlo* was configured with two objectives: maximizing throughput (#1) and minimizing cost (#2). We initially allowed only one server to participate in deployment (i.e., *Carlo* #1, *Carlo* #2, MTP, MinStage), and observed a performance decrease in the schemes MTP and MinStage. For comparison, we then allowed two servers to participate in deployment via a simple load-balancing approach (i.e., MTP w/2 and MinStage w/2). Since *Carlo* with objective #1 tended to deploy more INC applications on the data plane, resulting in less CPU resource usage than objective #2 (Fig. 7(b)). On the other hand, without the cross-plane collaboration deployment strategy, MTP and MinStage failed to implement all applications in the data plane. For fairness, we deployed the remaining applications on server(s) for them. Thus, because deployments exceed the capacity of server resources, both methods experienced a performance decrease when only one deployable server resource was available. Specifically, 11.0% and 15.4% of packet loss in MTP and MinStage are observed in Fig. 7(c), and compared to *Carlo* #1, the throughput of MTP and MinStage represents decreases of approximately 6.6% and 9.6%, respectively, in Fig. 7(d). This phenomenon was alleviated when we added the other available servers as deployment resources (i.e., MTP#2 and MinStage#2). In conclusion, *Carlo* can provide resource-aware and reasonable deployment solutions to avoid performance degradation.

(Exp#6) Performance of decision trees in testbed experiments. In this experiment, we implemented a decision tree algorithm and replayed the real dataset to evaluate how packet loss affects accuracy. In the switches, we collected flow features (e.g., packet length, and flag counter [6]) and used Digest [4] to send them to the control plane. The dataset includes four attack types: infiltration, web attack brute force (BF), SQL injection (SQL), and cross-site scripting (XSS).

The results in Table VI show the precision, recall, and F1 score for different attack types under three deployment methods. Due to packet loss in both MTP and MinStage, their precision scores decreased slightly across most attack types compared to Carlo. For example, for the Infiltration attack, Carlo achieves a precision of 0.783, slightly higher than MTP (0.757) and MinStage (0.75). This can be attributed to the small distribution of anomalous flows in the dataset. Notably, recall and F1 score for SQL attacks in MinStage dropped sharply (e.g., 14.2% in recall and 25% in F1 score), whereas *Carlo* maintained these metrics at 28.5% and 44.4%, respectively. This significant degradation in MinStage is mainly caused by packet loss during SQL flow collection, which severely limits detection capability. In contrast, *Carlo* consistently outperformed other methods across all metrics and attacks. This demonstrates that optimal deployment with *Carlo* preserves the effectiveness of INC applications.

IX. Conclusion

This paper proposed *Carlo*, a cross-plane deployment optimization framework for multiple INC applications. *Carlo* analyzes the resource requirements of diverse INC applications, formulates an ILP model to capture cross-plane constraints, and integrates heuristic and learning-based algorithms to balance solution quality and computational time. Compared to existing baseline methods, *Carlo* demonstrates superior deployment efficiency, achieving an average of 18.7% cost reduction, and ensuring stable application performance.

This work focuses on initial deployment optimization and does not currently include explicit failure recovery mechanisms. In the event of a switch or server failure, *Carlo* reruns its placement optimization, excluding the failed resources, to redeploy the affected applications. Integrating failure-aware redundancy or automated redeployment is a valuable direction for future work. Additionally, while *Carlo* evaluates on

TABLE VI: Performance of decision trees in testbed experiments.

Methods \ Attacks	Precision				Recall				F1 score			
	Infiltration	BF	SQL	XSS	Infiltration	BF	SQL	XSS	Infiltration	BF	SQL	XSS
Carlo	0.783	0.824	1.0	0.676	0.805	0.743	0.285	0.680	0.794	0.781	0.444	0.678
MTP	0.757	0.816	1.0	0.676	0.694	0.706	0.285	0.679	0.724	0.757	0.444	0.677
MinStage	0.75	0.814	1.0	0.666	0.66	0.697	0.142	0.650	0.705	0.751	0.25	0.658

Intel Tofino switches and servers, extending its deployment optimization to other P4 data plane targets (e,g, NetFPGA, SmartNICs) is feasible. The core framework is generalizable since it models application logic as CFGs and optimizes placement based on resource constraints. However, adaptation may require incorporating target-specific resource models and compilation constraints. Thus, Exploring the deployment of border heterogeneous programmable devices is also an important direction for future work.

ACKNOWLEDGMENTS

This work is partially supported by National Natural Science Foundation of China (NSFC) under Grant 62172189 and 62272050; the Guangdong Key Lab of AI and Multi-modal Data Processing, United International College (UIC) under Grant 2020KSYS007; the Guangdong Basic and Applied Basic Research Foundation under Grant No. 2021B1515120048; the Innovate UK grants 10098627, 10106629 and 600648; and the Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai).

REFERENCES

[1] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys & Tutorials*, 2022.

[2] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[3] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Transactions on Network and Service Management*, vol. 18, no. 4, pp. 4353–4364, 2021.

[4] "Intel Tofino switch ASIC," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, accessed on: Jul. 10, 2025.

[5] T. Ji, R. Vardekar, B. Vamanan, B. E. Stephens, and A. Akella, "MTP: Transport for in-network computing," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI), Philadelphia, PA*, 2025.

[6] C. Zheng, M. Zang, X. Hong, L. Perreault, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Planter: Rapid prototyping of in-network machine learning inference," *ACM SIGCOMM Computer Communication Review*, 2024.

[7] M. Jose, K. Lazri, J. François, and O. Festor, "Stateful InREC: Stateful in-network real number computation with recursive functions," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 830–845, 2022.

[8] B. Steinert, M. Häberle, J.-O. Nick, D. Farinacci, and M. Menth, "P4-lisp: A p4-based high-performance router for the locator/identifier separation protocol," in *IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 89–97.

[9] T. Xu, X. Wang, C. Tian, Y. Xiong, Y. Lin, and B. Ye, "CLIP: Accelerating features deployment for programmable switch," in *IEEE International Conference on Communications*. IEEE, 2023.

[10] F. Pereira, G. Matos, H. Sadok, D. Kim, R. Martins, J. Sherry, F. M. Ramos, and L. Pedrosa, "Automatic generation of network function accelerators using component-based synthesis," in *Proceedings of the Symposium on SDN Research*, 2022, pp. 89–97.

[11] X. Chen, H. Liu, D. Zhang, Q. Huang, H. Zhou, C. Wu, and Q. Yang, "Eliminating control plane overload via measurement task placement," *IEEE/ACM Transactions on Networking*, 2022.

[12] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "Compiling service function chains via fine-grained composition in the programmable data plane," *IEEE Transactions on Services Computing*, 2023.

[13] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015, pp. 103–115.

[14] W. Xu, Z. Zhang, Y. Feng, H. Song, Z. Chen, W. Wu, G. Liu, Y. Zhang, S. Liu, Z. Tian *et al.*, "ClickINC: In-network computing as a service in heterogeneous programmable data-center networks," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 798–815.

[15] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "TEA: Enabling state-intensive network functions on programmable switches," in *Proceedings of the ACM SIGCOMM*, 2020, pp. 90–106.

[16] X. Chen, Q. Huang, P. Wang, Z. Meng, H. Liu, Y. Chen, D. Zhang, H. Zhou, B. Zhou, and C. Wu, "LightNF: Simplifying network function offloading in programmable networks," in *IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021, pp. 1–10.

[17] J. Ma, S. Xie, and J. Zhao, "Flexible offloading of service function chains to programmable switches," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1198–1211, 2022.

[18] X. Zhang, L. Cui, W. Lau, F. P. Tso, Y. Deng, and W. Jia, "Carlo: Cross-plane collaboration for multiple in-network computing applications," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024.

[19] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Hybrid in-network classification using programmable switches," *IEEE/ACM Transactions on Networking*, 2024.

[20] R. MacDavid, X. Chen, and J. Rexford, "Scalable real-time bandwidth fairness in switches," *IEEE/ACM Transactions on Networking*, 2023.

[21] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time DDoS attack detection to programmable data planes," in *IFIP/IEEE Symposium on Integrated Network and Service Management*. IEEE, 2019, pp. 19–27.

[22] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 8–14.

[23] "Intel corporation. Data Plane Development Kit." https://www.dpdk.org/, accessed on: Jul. 10, 2025.

[24] X. Chen, H. Liu, Q. Huang, P. Wang, D. Zhang, H. Zhou, and C. Wu, "SPEED: Resource-efficient and high-performance deployment for data plane programs," in *IEEE 28th International Conference on Network Protocols (ICNP)*. IEEE, 2020, pp. 1–12.

[25] T. Li, Z. Wei, X. Chen, and Z. Zhu, "MrgRecmp4: Efficient stateful SFC recompilation in PDP switches with flexible table merging," in *IEEE Conference on Computer Communications (IEEE INFOCOM)*. IEEE, 2025.

[26] X. Zhang, L. Cui, F. P. Tso, Z. Li, and W. Jia, "Dapper: Deploying service function chains in the programmable data plane via deep reinforcement learning," *IEEE Transactions on Services Computing*, 2023.

[27] T. Wang, Q. Fan, C. Wang, L. Yang, L. Ding, N. J. Yuan, and H. Xiong, "FlagVNE: a flexible and generalizable reinforcement learning framework for network resource allocation," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2024, pp. 2406–2414.

[28] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore, "Machine-learning-assisted DDoS attack detection with P4 language," in *IEEE International Conference on Communications*. IEEE, 2020, pp. 1–6.

[29] J. Hyun, N. Van Tu, and J. W.-K. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.

[30] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *30th USENIX Security Symposium*, 2021, pp. 3829–3846.

[31] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow," in *USENIX Annual Technical Conference*, 2018, pp. 823–835.

[32] E. Cidon, S. Choi, S. Katti, and N. McKeown, "Appswitch: Application-layer load balancing within a software switch," in *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017, pp. 64–70.

[33] N. Van Tu, J. Hyun, G. Y. Kim, J.-H. Yoo, and J. W.-K. Hong, "INT-collector: A high-performance collector for in-band network telemetry," in *14th International Conference on Network and Service Management*. IEEE, 2018, pp. 10–18.

[34] S. Tang, D. Li, B. Niu, J. Peng, and Z. Zhu, "Sel-INT: A runtime-programmable selective in-band network telemetry system," *IEEE transactions on network and service management*, vol. 17, no. 2, pp. 708–721, 2019.

[35] S. Patel, R. Atsatsang, K. M. Tichauer, M. H. Wang, J. B. Kowalkowski, and N. Sultana, "In-network fractional calculations using P4 for scientific computing workloads," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 33–38.

[36] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in P4," in *IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[37] Y.-K. Lai, S.-Y. Yu, I.-S. Chan, B.-H. Huang, C.-H. Chang, J. H. Chen, and J. Mambretti, "Sketch-based entropy estimation: a tabular interpolation approach using P4," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 57–60.

[38] V. S. Thapeta, K. Shinde, M. Malekpourshahraki, D. Grassi, B. Vamanan, and B. E. Stephens, "Nimble: Scalable TCP-friendly programmable in-network rate-limiting," in *Proceedings of the ACM SIGCOMM Symposium on SDN Research*, 2021, pp. 27–40.

[39] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *32nd USENIX Security Symposium (USENIX Security)*, 2023, pp. 6203–6220.

[40] "Perf WiKi," https://perf.wiki.kernel.org, accessed on: Jul. 10, 2025.

[41] I. S. Association *et al.*, "IEEE standard for ethernet amendment 3: Physical layer specifications and management parameters for 40 Gb/s and 100 Gb/s operation over fiber optic cables," 2015.

[42] "Gurobi - The Fastest Solver," https://www.gurobi.com/, accessed on: Jul. 10, 2025.

[43] "Open AI Spinning Up," https://spinningup.openai.com/en/latest/, accessed on: Jul. 10, 2025.

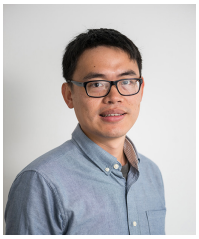
[44] Y. Yoshinaka, J. Takemasa, Y. Koizumi, and T. Hasegawa, "On implementing ChaCha on a programmable switch," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 15–18.

[45] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization." *ICISSp*, vol. 1, pp. 108–116, 2018.

Xiaoquan Zhang is currently a postdoctoral researcher in the College of Cyber Security at Jinan University, Guangzhou, China. He received the Ph.D. degree from Jinan University, China, in 2024. His current research interests include programmable data planes, software-defined networking, and machine learning in computer networks.

Lin Cui is currently a professor in the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on programmable data plane, intelligent networks, in-network computing, software defined networking (SDN), congestion control, and cloud data center networking.

Waiming Lau is currently a postgraduate student in the Department of Computer Science at Jinan University, Guangzhou, China. He received the B.Eng. degree from Jinan University in 2021. His research interests include Software-Defined Networks, Routing, and Congestion Control.



Fung Po Tso received his B.Eng., M.Phil. and Ph.D. degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently a professor in the Department of Computer Science at the Loughborough University. His research interests include: His research interests include: Network resource management, edge computing, edge AI, In-network computing, digital twins, and cybersecurity.

Yuhui Deng received the Ph.D. degree in computer science from the Huazhong University of Science and Technology in 2004. He is currently a Professor with the Computer Science Department, Jinan University. He worked with the EMC Corporation as a Senior Research Scientist from 2008 to 2009. He worked as a Research Officer with Cranfield University, UK, from 2005 to 2008. His research interests cover green computing, information storage, computer architecture, and performance evaluation.

Weijia Jia is currently a Chair Professor at Beijing Normal University (BNU Zhuhai), and BNBU, Zhuhai, China. His research interests include smart city, IoT, knowledge graph constructions, multicast and anycast QoS routing protocols, wireless sensor networks, and distributed systems. He has over 500 publications in prestigious international journals/conferences and research books and book chapters.